Computer Science Department

TECHNICAL REPORT

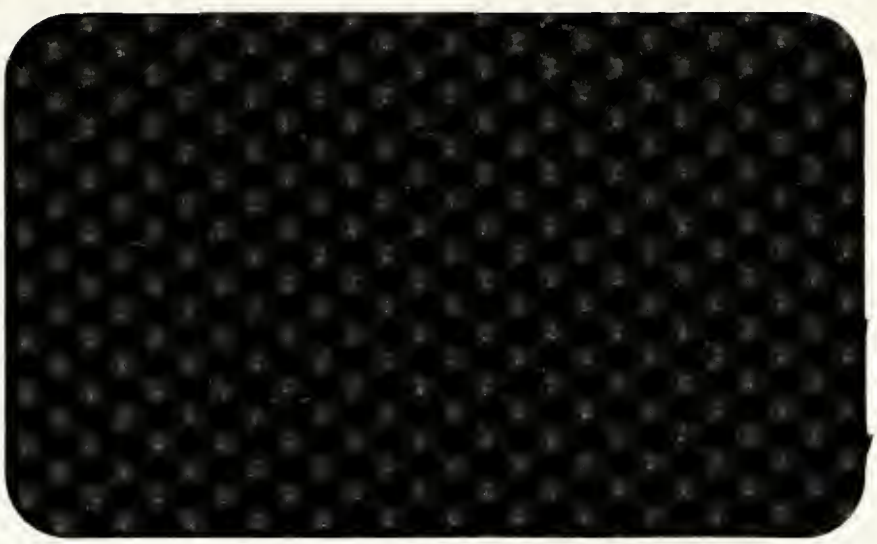
THEORIES OF PROGRAM TESTING AND THE APPLICATION OF REVEALING SUBDOMAINS

By
Elaine J. Weyuker
and
Thomas J. Ostrand

February 1979
Report No. 008

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012



THEORIES OF PROGRAM TESTING AND THE APPLICATION OF REVEALING SUBDOMAINS

By
Elaine J. Weyuker
and
Thomas J. Ostrand

February 1979
Report No. 008

Theories of Program Testing and the Application of Revealing Subdomains

Elaine J. Weyuker
Courant Institute of Mathematical Sciences
New York University
New York, New York

Thomas J. Ostrand
Software Research Group
Sperry Univac
Blue Bell, Pennsylvania

Theories of Program Testing and the Application of Revealing Subdomains

Elaine J. Weyuker

Thomas J. Ostrand

ABSTRACT

The theory of test data selection proposed by Goodenough and Gerhart is examined and a number of theoretical and pragmatic deficiencies are identified. The concepts of a revealing test criterion and a revealing subdomain are proposed to overcome some of these weaknesses, and to provide a realistic basis for a theory of testing.

A subset of a program's input domain is revealing if the existence of one incorrectly processed input implies that all the subset's elements are processed incorrectly. The intent of this notion is to partition the program's domain in such a way that all elements of an equivalence class are either processed correctly or incorrectly. A program test set is then formed by choosing one element from each class. The technique is especially useful for programs whose computations depend on a classification of their input domain, but it can also be useful for programs which process their entire domain uniformly.

A methodology for forming revealing subdomain partitions is described, and illustrated with three examples to which other testing methodologies have been applied in the literature.

for uniform testing of any program intended to meet a given set of specifications.

In Section 5 we introduce the concepts of a revealing test selection criterion and a revealing subdomain. These definitions address some of the deficiencies in the Goodenough and Gerhart theory, and are intended to capture the idea of testing to exhibit the presence of errors, rather than to be assured of their absence. Section 6 presents a methodology for constructing revealing subdomains, and illustrates its application to several example programs.

2. The Goodenough and Gerhart Definitions

We begin by examining the Goodenough and Gerhart definitions for test criteria properties to see whether they have captured the necessary features. We use their notation: F is a program, D the input domain, and R the output domain for F. On input $d \in D$, F (if it terminates) produces output $F(d) \in R$. The output specification for F is given by OUT(x,y), where $x \in D$ and $y \in R$. F is correct on input d (abbreviated OK(d)) if F(d) exists and OUT(d,F(d)).

A test T for program F is simply a (finite) subset of D. A test selection criterion C is a predicate on subsets of D which selects certain sets of inputs as appropriate tests. The key definitions in [5] are the following: A test T is <u>successful</u> iff $(\forall t \in T)(OK(t))$. A test selection criterion C is <u>reliable</u> iff either every test selected by C is successful, or no test selected is successful. C is <u>valid</u> iff whenever program F is incorrect, then C selects at least one test set T which is not successful for F. From these definitions, the fundamental theorem of [5] follows: If C is a reliable and valid criterion, then any test selected by C is an ideal test. We shall call a criterion which is both reliable and valid for a program F, an

ideal criterion.

There are several serious deficiencies in the theory founded on this theorem. First is that the concepts of reliability and validity are defined with respect to the entire input domain of a program, and thus allow no discrimination among errors. A criterion which exposes only a single error is valid regardless of the number of errors present in the program in question. We discuss a way to avoid this problem in Section 5.

The second difficulty comes about because all the above definitions are relative to a single program. A criterion which is reliable or valid for F is not necessarily so for F'. The problems arising with tests based solely on the structure of a program are avoided in [5], but the tests are nevertheless based entirely on the behavior of the single program under consideration.

Another problem is the lack of independence of the properties of validity and reliability. This problem will be discussed in Section 3, where we show how it adversely affects the theory's practical utility.

Fourth, neither validity nor reliability is preserved throughout the debugging process. That is, a criterion which is valid at the start of debugging does not necessarily remain so for every intermediate program produced by successive corrections made on the way to an error-free program. Furthermore, an invalid criterion may become valid as the program is transformed. The same is true for reliability.

From the point of view of the programmer who has just designed a program or a person assigned the task of testing and debugging someone else's program, it is practically impossible to find test selection criteria which are both reliable and valid, since these properties depend on the nature of errors present in the program. The situation is illustrated with several simple examples.

First, if program F is correct, i.e., if $(\forall d \in D)(OK(d))$, then any test will be successful and every selection criterion C is reliable and valid. Of course, F is not known to be correct, and the programmer must use other means to show the reliability and validity of C. Such other means will be the equivalent of a proof of the program's correctness.

On the other hand, if F is not correct, there is no way of knowing whether a criterion is ideal without knowing the errors in F. The first example in [5] illustrates this point. The program F computes d*d, for d an integer, while the output specification is F(d) = d+d. Since F is correct for d = 0 and d = 2, and incorrect for all other inputs, a criterion which selects as tests only subsets of {0,2} is reliable but not valid, since it does not indicate the error in F. A criterion which selects subsets of {0,1,2,3,4} exposes the error in F, and is therefore valid, but is not reliable since T = {0,2} is a successful test, while T = {0,1} is not successful.

A slight change in the program, while retaining the same output specification, completely changes the reliability and validity of these criteria. If F' computes d+2, then d = 2 is the only input for which a correct answer is produced. Thus, choosing subsets of {0,2} becomes a valid, but not reliable criterion for F'. If F'' computes d+5, the criterion which selects subsets of {0,1,2,3,4} is now both reliable and valid.

The problem facing the tester is to find an ideal criterion for his program. However, as the above examples show, there is no way to know if a criterion is ideal without already knowing the errors in the program, surely a paradoxical situation.

3. Practical Testing Considerations

We now examine the usefulness of an ideal test for the practical problems

of program writing and debugging. Typically, a program goes through many different versions, steps of refinement, improvements, modifications, etc., as it is being written. It is desirable to verify the correctness of each stage in order to increase confidence in the correctness of the next one. An ideal test criterion for one version is not necessarily an ideal criterion for another. The debugging process constantly changes the program being worked on; errors are located and removed, and sometimes new errors are inadvertently introduced.

Consider the following modification of the example presented earlier:

$$F(d) = (d*d) + 3$$

$$OK(d) = [(d+d) + 6 = F(d)]$$

That is, F is supposed to double the integer d and add 6, but instead F squares d and adds 3.

Note that OK(3) and OK(-1), but ~OK(d) for all other d. Thus, $C(T) = (T = \{t\}$ and $t \in \{0,1,2\})$ is a valid and reliable criterion. Now suppose that by running F on one of the input values of T, the error of adding the wrong constant is located. This error is corrected, and now F(d) = (d*d) + 6. At this point, OK(0) and OK(2), but ~OK(d) for all other d. Hence C is no longer a reliable criterion, since some tests satisfying C will detect the remaining error, and others will not.

Note that the reverse situation can also occur. Thus, for example, $C'(T) = (T = \{t\}$ and $t \in \{-1,1,3\})$ is valid but not reliable for the original program containing two errors. For the partially corrected program, however, it is both valid and reliable.

Thus properties of criteria are not maintained throughout the debugging phase, nor are they even "monotonic" in the sense of being either uniformly gained or preserved, or uniformly lost or preserved.

Are reliabilty and validity properties which testers should attempt to give their tests? The authors of [5] point out the unreliability of test selection criteria based solely on internal program structure and state that the success of tests based on an unreliable criterion is poor evidence that a program contains no errors.

In practice, however, it seems extremely unlikely that any reasonable test selection criterion would be reliable in the Goodenough and Gerhart sense. Typical tests or groups of tests are constructed to check the possibility of various types of errors which may exist. The criterion which selected these tests would be reliable only if the program contained either no errors, or all the possible errors. (An additional possibility is that the criterion is so poorly designed that it exposes none of the errors which do exist.) Obviously most programs will be somewhere in between the first two cases, and we should not expect all test results to be the same. The third case would arise only if the criterion were invalid, clearly the worst type of test.

It is interesting to note that the Goodenough and Gerhart notions of reliability and validity are not independent; at least one of the two properties must hold for any criterion.

Theorem 1: $(\forall C)\ (Val(C) \vee Rel(C))$

This is true because an invalid test criterion exposes no errors, and therefore all its tests are successful. Thus, possession of either reliability or validity alone says nothing about the quality of a test selection criterion. In fact, a way to guarantee validity is to use an unreliable criterion, and as we saw above, most criteria probably are

unreliable.

4. Uniformly Valid and Reliable Test Criteria

We have seen that Goodenough and Gerhart's properties for ideal tests suffer from a fault similar to program structure-based tests, namely, dependence on the particular program being tested. A possible remedy for the situation is to look for tests whose validity and reliability are dependent only on the desired output specification. Such a test would be universal for any program written to satisfy the given specification. Note that in the definitions of validity and reliability in [5] there is an implicit free variable F, referring to the particular program under consideration. The formal definitions of uniform validity and reliability given below bind this F with an outermost universal quantifier. Note that we have also modified the notation of [5] for SUCC and OK, by including the program F as an explicit parameter.

Given an input domain D and output specification OK(F,d), criterion C is uniformly valid iff

$$(\forall F)[(\exists d \in D)(\neg OK(F,d)) \supset (\exists T \subseteq D)(C(T) \wedge \neg SUCC(F,T))].$$

Criterion C is uniformly reliable iff

$$(\forall F)(\forall T_1, T_2 \subseteq D)[(C(T_1) \wedge C(T_2)) \supset (SUCC(F,T_1) \equiv SUCC(F,T_2))].$$

A uniformly ideal test selection criterion for a given output specification is both uniformly valid and reliable. Such a definition would solve all the program dependent difficulties inherent in the definitions of [5]. Unfortunately, however, the concept of a uniformly ideal criterion also has serious flaws. Most important, for any significant program there can be no uniformly ideal criterion which is not in a sense trivial. To see this, let us call a criterion C trivially valid if the union of all tests selected

by C is D. Obviously a trivially valid criterion is valid. However, a criterion C which is not trivially valid cannot be uniformly valid for a given output specification, since for any element d not included in any of C's tests, a program can be written which is incorrect for d, and correct for D - {d}. Thus we have:

Theorem 2: A criterion C is uniformly valid if and only if C is trivially valid.

A similar weakness holds for uniform reliability.

Theorem 3: A criterion C is uniformly reliable if and only if C selects a single test.

Proof: If C selects only one test it is clearly reliable for any program. Suppose C selects different tests T_1 and T_2, and that t is an input in T_1, but not in T_2. A program F exists which is correct on all the inputs in T_2, but incorrect on t. The two tests thus yield different results for F, and C is not reliable. □

Let us now consider the uniform analogue of the fundamental theorem of [5]. This theorem states that if C is a reliable and valid criterion for a given program F, and if F runs correctly on every element of a test which satisfies C, then F runs correctly on every element in the domain. In contrast to this, our next result shows that there can be no criterion which uniformly reveals errors.

Theorem 4: $(\forall C)(\forall T \subsetneq D)[\, C(T) \supset (\exists F)(SUCC(F,T) \wedge \sim SUCC(F,D))]$

Proof: Since there is some input datum $t \notin T$, let F be a program which is correct on T and which does not halt on t. □

This result states essentially that no matter what criterion is used to select tests for a given specification, and no matter what test is chosen (if it is not all of D), there will always be a program which can defeat the test. This is another way of expressing the oft quoted statement that "testing can only reveal the presence of errors, never their absence" [3].

Other similar negative results have been proved. Weyuker [12] has shown that there can be no algorithm which can decide whether or not a given statement, branch, or path of a program may ever be exercised, nor whether or not every such unit may be exercised. Thus, a testing methodology which requires the generation of data to do one or more of these things cannot be guaranteed to terminate.

Howden [7] showed that there is no procedure which, given an arbitrary program F and output specification, will produce a non-empty finite test set $T \subseteq D$ such that if F is correct on T then F is correct on all of D. The reason behind this result is that the non-existent procedure is expected to work for all programs, and we thus encounter familiar non-computability limitations. Howden's response is to look for subclasses of programs for which test-generating procedures can be successful.

Similarly, in practical terms, one does not have to design test criteria to deal with every possible incorrect program. Programmers do not write pathologically wrong programs, but rather attempt to meet the desired output specifications. A theory of testing should attempt to take advantage of classes of errors which are known to occur frequently, and should capture the essence of test selection criteria which can expose the presence of such errors or guarantee their absence.

5. Revealing Test Selection Criteria

The theory proposed in [5] shows that testing can establish the absence of errors in a program, if the test selection criteria satisfy the right properties. Both the theoretical problems and the difficulties which arise in its application, however, render the theory of little pragmatic use. As we saw in Sections 2 and 3, finding valid and reliable criteria for real programs is all but impossible. Thus, although theoretically testing can be used to show the absence of errors, pragmatically it remains true that testing can only expose their presence.

To provide an effective foundation for testing real programs, a theory of testing must characterize tests which uncover errors. In addition, a single test or test selection criterion must not be expected to do everything. Different types of errors are exposed by different test methods; different parts of a program's input domain may be subject to different errors. A usable theory of testing must permit a test set to contain both correctly and incorrectly processed inputs.

The notions of revealing test criterion and revealing subdomain are intended to capture these properties of tests. The definitions overcome some, but not all the weaknesses of the theory in [5]. In the following section we give several examples of programs and revealing subdomains constructed for them. In practice, it is important to use information from both program-based and program-independent sources. The approach we use combines the techniques of path domain testing [7], functional testing [9], and special values testing [6].

Formally, a test criterion C is revealing for a subset S of the input domain if whenever S contains an input which is processed incorrectly, then

every test set which satisfies C is unsuccessful. Equivalently, if any test selected by C is successfully executed, then every input in S produces correct output. A formal definition is

$$\text{REVEALING}(C,S) \triangleq (\exists d \in S)\ (\sim OK(D)) \supset (\forall T \subseteq S)\ (C(T) \supset \sim SUCC(T))$$

The property of revealing may seem like a very strong requirement to put on a test selection criterion, but the domain subset parameter allows enough precision to make the definition useful. Any test selected by a criterion which is revealing for subdomain S is an ideal test for S.

Two special cases of revealing criteria are worth noting. The first is when the subdomain S is the entire input domain D of the program. In this case the property revealing is equivalent to the conjunction of validity and reliability, so long as vacuous cases are excluded.

Theorem 5: A non-empty criterion C is revealing with respect to the domain D of a program if and only if C is both valid and reliable.

Proof: Suppose C is revealing. If there are any errors in D then every test meeting C must be unsuccessful. C is therefore reliable, and since it is non-empty, it is also valid.

If C is both valid and reliable, then the presence of any error guarantees that every test meeting C will be unsuccessful.□

Our characterization of a good test criterion is thus coextensive with that of Goodenough and Gerhart, although as noted earlier, the attempt to cover all possible errors with a single test criterion is unlikely to be successful.

The second special case occurs when the criterion allows any subset of S as a test case. The two parameters then become one, and we say that a subdomain S is revealing if the occurrence of an error is S implies the failure of any subset of S. In other words, there is an error in S if and only if all inputs in S produce errors. The conjunction of these two special cases implies that either the program is correct or the program processes every input incorrectly. Clearly this is not likely to occur frequently, nor will we know when it does occur. It is not of pragmatic interest.

To make use of a property as strong as the second case essentially requires that the problem's input domain be partitioned in an intelligent and meaningful way. We discuss several examples below, and give guidelines for choosing appropriate subdomains.

If we have a revealing criterion for a subdomain S of D, running successful tests based on that criterion only assures us of the correctness of our program on S. Generally, one should look for criteria which are revealing on subdomains whose union is all of D. Of course, if a particular problem or program is amenable to testing for part of its domain and, for example, formal verification for another part, then both methods should be used.

In a study of symbolic testing and its effectiveness in detecting errors [8] , Howden states that "both symbolic testing and actual data testing are unreliable for discovering most of the path-domain errors in the sample programs.... Actual data testing can reveal the error if the programmer is lucky enough to choose a test which comes from the incorrect part of the domain."[1]

The notion of a revealing subdomain is a partial solution to the problem described above. If a program's input domain can be successfully divided into

---

1. [8], p. 274.

revealing subdomains, then it is no longer necessary to rely on luck in choosing appropriate tests. Either every element of a subdomain produces the correct output, or none does.

6. Constructing Revealing Criteria

Test data can be developed from both the program under consideration and from program-independent sources such as the problem's specifications, the algorithm used, and the input/output data structures. In attempting to find revealing criteria or subdomains for a program, we start by considering the program separately from the program-independent sources.

The input domain is first partitioned into path domains. A path domain consists of a set of inputs which each follow the same path, or one of a family of related paths, through the program's flow graph. A second preliminary partition, the problem partition, is formed on the basis of common properties implied by the specifications, algorithm, and data structures. These two partitions are then intersected to form classes which are the basis for test selection.

For example, a problem whose single input is an integer might have specifications which suggest a partition into the odd and even integers. Analysis of the program graph may show that all positive inputs follow one path, all negative inputs another, and zero is processed as a special case. Intersecting the problem partition with the path domain partition results in a set of five classes: odd positive integers, odd negative, even positive, even negative, and zero. If our understanding of the problem is good, these classes are likely to be revealing for many errors.

Usually the path domain partition does not differ so markedly from the problem partition, since ultimately the program, data structures, and algorithm all derive from the original problem specifications. The differences which do exist are fruitful places to look for errors. This is the basic rationale behind our method of forming subdomains.

Informally, the process attempts to construct a partition of the input domain such that elements in an equivalence class are processed the same way by the program, and in addition are characterized the same way by the specifications and the algorithm. The classes of such a partition are usually revealing for most errors.

An error for which not every subdomain is revealing is usually the result of a pathological special case in the computation. Frequently, most of the subdomains are revealing for such errors. Note that to detect an error by testing, it is only necessary that one subdomain be revealing and produce incorrect output due to that error. Examples of this type of error are given for the exponentiation program (Example 2).

We now give several examples of programs and revealing subdomain construction. The actual test data consists simply of an arbitrary element from each subdomain.

Example 1: This is a triangle classification problem which has been studied by several authors [2,11]. The problem's specifications are

Input: Three positive numbers A,B,C; with $A \geq B \geq C$

Output: The program indicates which of the following descriptions is satisfied by A, B, and C.

1. They cannot be the sides of any triangle
2. They are the sides of an equilateral triangle
3. They are the sides of an isosceles, but not equilateral, triangle
4. They are the sides of a scalene right triangle
5. They are the sides of a scalene obtuse triangle
6. They are the sides of a scalene acute triangle

Figure 1 shows the program presented in [2] for the triangle classification.

This problem is especially suitable for the application of the revealing subdomain methodology. Since the very purpose of the program is to classify its input domain, there is an obvious specification-based partition. In addition, since there are no loops in the program, there are finitely many path

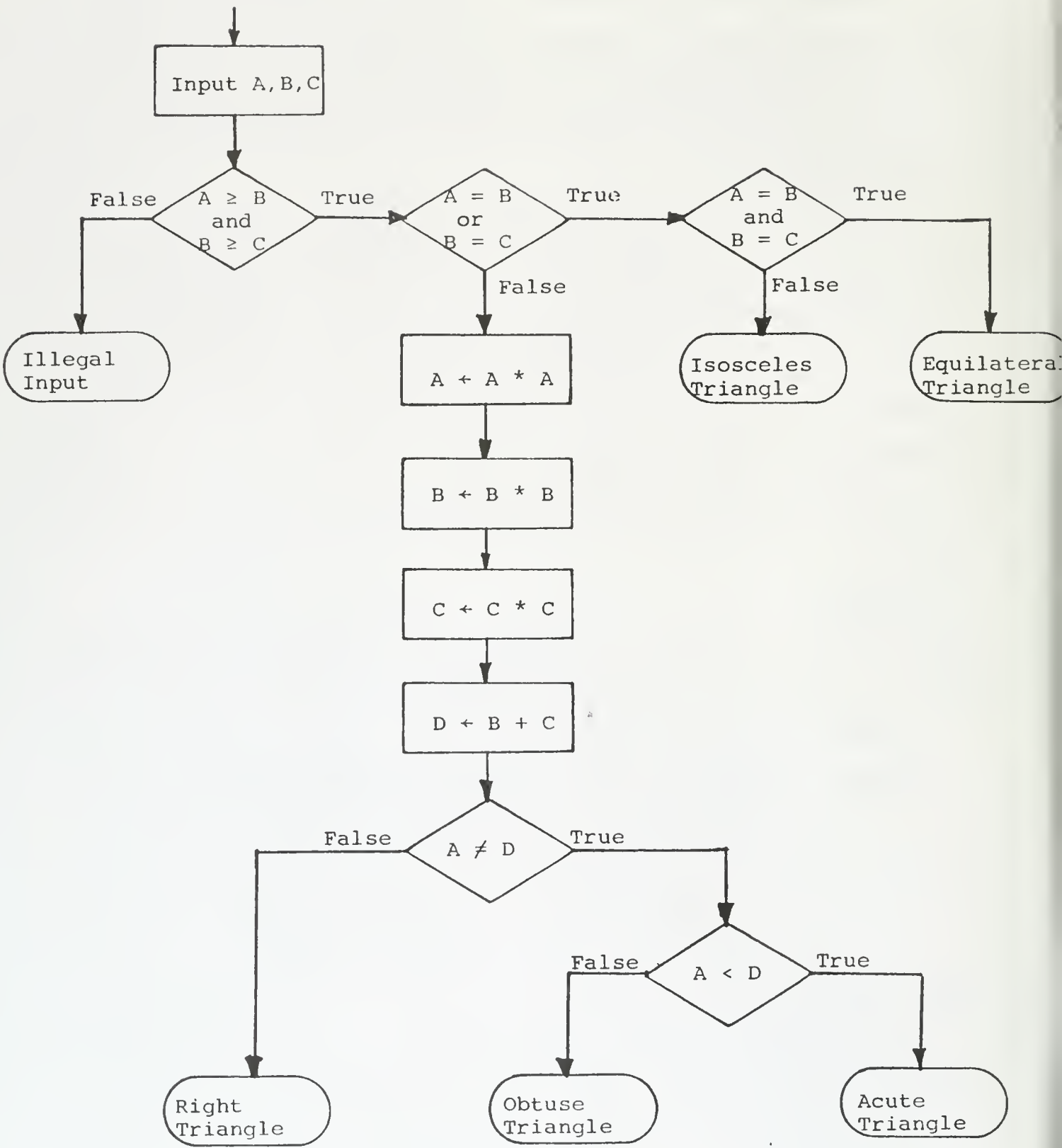


Figure 1. Flowchart for triangle classification problem

domains. The resulting subdomains (constructed by intersecting the problem partition and the path domain partition) therefore separate all significant cases for this problem, and we can expect them to be revealing for practically any type of error.

Each path domain for the triangle program is described simply by the conjunction of the predicate branches taken on the given path. Conditions describing the six path domains, P_1 - P_6, appear in Figure 2.

To form the specification-based partition, we first divide the universe of triples of positive numbers into legal and illegal input forms. S_1 is all triples such that $\sim(A \geq B \geq C)$ or equivalently $(A < B) \vee (B < C)$; these are the illegal inputs. For the legal inputs we now distinguish two conditions. S_2 contains all triples such that $A \geq B \geq C$ and $A \geq B+C$; in this case the triangle inequality is not satisfied, and A,B,C cannot be the sides of a triangle ($B \geq A+C$ or $C \geq A+B$ cannot occur, since $A \geq B \geq C$). Inputs such that $A \geq B \geq C$ and $A < B+C$ are legally ordered and represent valid triangles. These triples are further divided into six subclasses:

S_3:	$A = B = C$	(equilateral)
S_4:	$A = B > C$	(isosceles)
S_5:	$A > B = C$ and $A < B+C$	(isosceles)
S_6:	$A > B > C$ and $A^2 = B^2+C^2$	(right scalene)
S_7:	$A > B > C$ and $A^2 < B^2+C^2$	(acute scalene)
S_8:	$A > B > C$ and $A^2 > B^2+C^2$ and $A < B+C$	(obtuse scalene)

Intersecting the six path domains with the eight specification domains results in the nine non-empty subdomains D_1 - D_9 in Figure 3. These are the basis for the test data used for the triangle program. Figure 4 shows a test

P_1: $\sim[(A \geq B) \wedge (B \geq C)] \equiv (A < B) \vee (B < C)$

P_2: $(A > B > C) \wedge (A^2 = B^2+C^2)$

P_3: $(A > B > C) \wedge (A^2 > B^2+C^2)$

P_4: $(A > B > C) \wedge (A^2 < B^2+C^2)$

P_5: $(A \geq B \geq C) \wedge [(A = B) \vee (B = C)] \wedge \sim[(A = B) \wedge (B = C)]$
$\equiv (A = B > C) \vee (A > B = C)$

P_6: $(A = B = C)$

Figure 2. Path domain conditions for triangle classification program

$D_1 = S_1 \cap P_1$: $(A < B) \vee (B < C)$

$D_2 = S_2 \cap P_3$: $(A \geq B+C) \wedge (A > B > C)$

$D_3 = S_2 \cap P_5$: $(A \geq B+C) \wedge (B = C)$

$D_4 = S_3 \cap P_6$: $(A = B = C)$

$D_5 = S_4 \cap P_5$: $(A = B > C)$

$D_6 = S_5 \cap P_5$: $(A > B = C) \wedge (A < B+C)$

$D_7 = S_6 \cap P_2$: $(A > B > C) \wedge (A^2 = B^2+C^2)$

$D_8 = S_7 \cap P_4$: $(A > B > C) \wedge (A^2 < B^2+C^2)$

$D_9 = S_8 \cap P_3$: $(A > B > C) \wedge (A^2 > B^2+C^2) \wedge (A < B+C)$

Figure 3. Subdomains formed by intersection of problem partition and path domain partition

item arbitrarily chosen from each subdomain, together with the program output and the correct output for each item. The program produces an incorrect result for subdomains D_2 and D_3. In both cases the error is that the program incorrectly reports a triple to be the sides of a valid triangle of a certain type. D_2 and D_3 are revealing for these errors; all their elements will be processed incorrectly in the same way.

Note that in [2] DeMillo, Lipton, and Sayward fail to detect these errors; one of their test cases incorrectly categorizes (14,6,4) as an obtuse triangle.

While our treatment of the triangle problem produces an effective set of test cases, it nevertheless fails to deal with a type of error which may be quite common in real programs. Note that the program in Figure 1 will report that (-3,-4,-5) are the sides of an acute triangle. However, this input and all other triples containing negative values are not included in any of the subdomains of Figure 3. Naturally, we developed these subdomains on the assumption that the inputs would all be positive numbers, but there is hardly more reason to make this assumption than to assume the inputs will be in non-decreasing order. It is necessary to differentiate between two types of invalid input situations. In the first the inputs simply are of the wrong form for the given problem, as with the negative triangle inputs. In the second the inputs have the proper form for the problem, but fail to satisfy some required property, such as the triangle inequality.

To expose the effects of a program's failure to respond properly to a wrong form type of input, a special subdomain can be established for all such inputs. The second type of invalid input should be treated similarly to valid inputs, since in general some processing beyond the input operations is needed to decide its invalidity.

Domain	Test Data	Correct Output	Actual Output
D_1	(1, 2, 3)	illegal order	illegal order
D_2	(14,6, 4)	not a triangle	obtuse
D_3	(2, 1, 1)	not a triangle	isosceles
D_4	(1, 1, 1)	equilateral	equilateral
D_5	(2, 2, 1)	isosceles	isosceles
D_6	(3, 2, 2)	isosceles	isosceles
D_7	(5, 4, 3)	right	right
D_8	(6, 5, 4)	acute	acute
D_9	(4, 3, 2)	obtuse	obtuse

Figure 4. Test data and output for triangle classification program

Example 2: This program is simple enough to be proved correct, and many people have done so.[1,10] The problem is to calculate x to the power y, where x is an integer and y is an non-negative integer. The method used (Figure 5) does the computation in time proportional to log y, by taking advantage of the binary representation of y.

We number the action boxes in the program's graph, and see that the set of paths through it is a subset of the regular set $1(3 \vee 23)^*$. The set of feasible paths for legal inputs is $1 \vee 1(3 \vee 23)^*23$, since the most significant bit in the binary representation of y must be a 1. Note that the path taken for input (x_o, y_o) depends only on the value of y_o.

Since there are infinitely many feasible paths, we cannot use test data from every path domain. We choose several domains which are representative of different types of y_o values; the paths and their associated y_o values are shown in Figure 6.

We now examine the problem specifications and the algorithm to find natural ways of subdividing the x and y domains. An obvious partitioning for x is based on whether x_o is positive, negative, or zero. From positive x_o values we single out 1, since it is easy for the output to be correct if $x_o = 1$, but incorrect otherwise. We similarly single out $x_o = -1$. The x inputs are thus partitioned into the five classes:

$$X_1: \quad x_0 < -1,$$
$$X_2: \quad x_0 = -1,$$
$$X_3: \quad x_0 = 0,$$
$$X_4: \quad x_0 = 1,$$
$$X_5: \quad x_0 > 1.$$

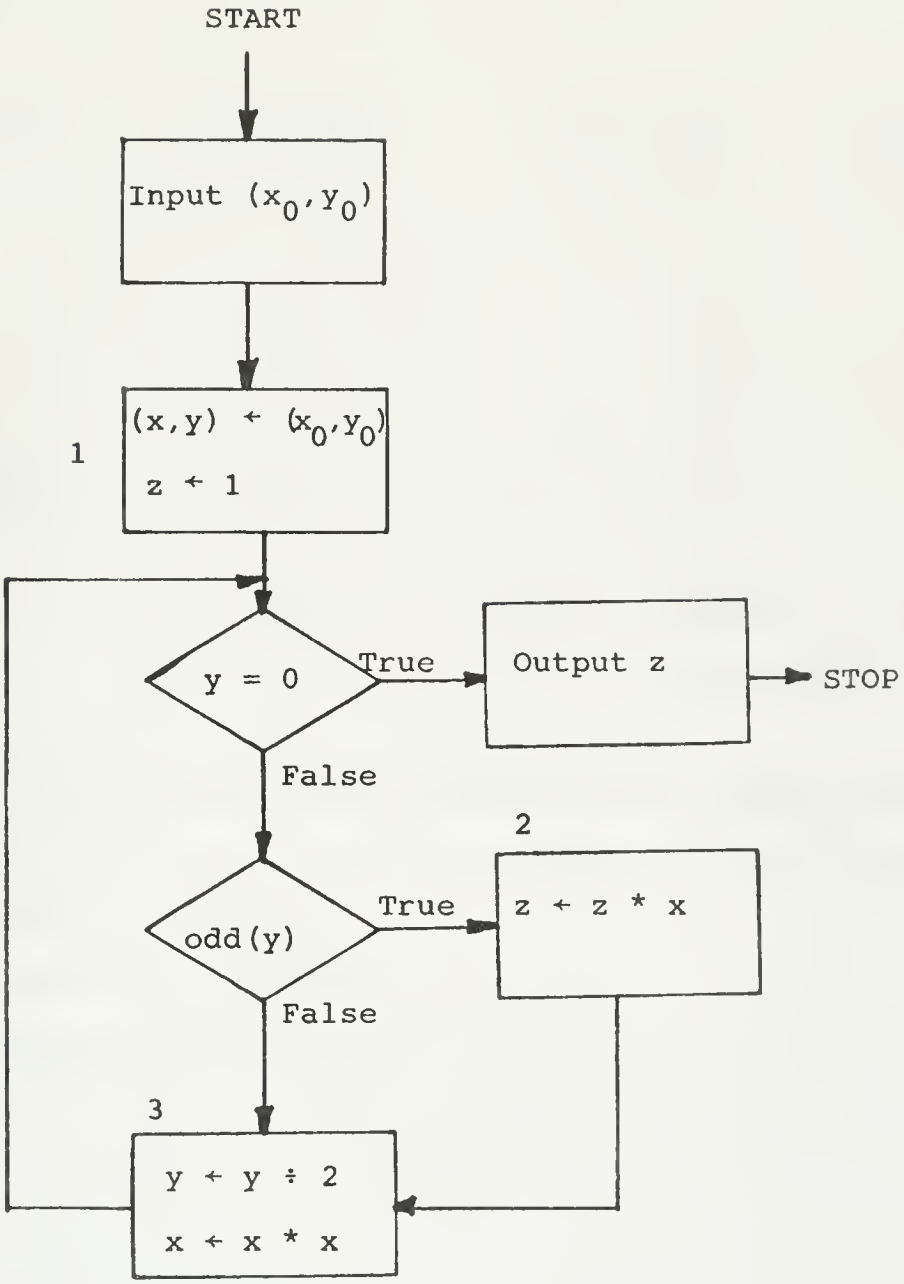


Figure 5. Flowchart for fast exponentiation program

Path Domain	Description of path	Binary form of y_0	Decimal value of y_0
Y_1	1	0	0
Y_2	13^n23, $n \geq 1$	10^n	2^n
Y_3	$1(23)^n$, $n \geq 1$	1^n	$2^n - 1$
Y_4	$1(323)^n$, $n \geq 2$	$(10)^n$	$\frac{2}{3}(4^n - 1)$
Y_5	$1(233)^n23$, $n \geq 1$	$(10)^n1$	$\frac{1}{3}(4^{n+1} - 1)$

Figure 6. Path domains of exponentiation program

The specifications and algorithm suggest similar natural classes for the exponent y. The three classes $y_o < 0$, $y_o = 0$, and $y_o > 0$ should be considered, as well as certain special cases when y_o is positive. The case where $y_o = 2^k$ is of special interest, since in this case the algorithm works by accumulating the entire power in x, and does not use z until the last loop iteration. Similarly $y_o = 2^k - 1$ is unusual, since the power is accumulated entirely in z. Note that these two special cases for y_o, as well as $y_o = 0$, were already identified as separate classes in the path domain analysis (See Figure 6).

Combining these specification- and algorithm-based classes with the path domain classes gives the 26 subdomains summarized in Figure 7. Since Y_1 represents illegal values for y, we can form a single subdomain where $y_o < 0$ and x_o is arbitrary.

We now examine some errors which could have been written into the code, and their effect on the program's output; our objective is to see for which errors the subdomains are revealing.

Error 1: z is initialized to 0 instead of to 1

Effect: The program's output is always 0. This is incorrect for all inputs except $x_o = 0$, $y_o > 0$. Thus every input in the subdomains built from X_1, X_2, X_4, and X_5 will produce the wrong output; inputs in $X_3 \times (Y_2, Y_3, Y_4, Y_5)$ produce correct output. The output is incorrect for the single member of $X_3 \times Y_1$. Every subdomain is revealing for Error 1.

Error 2: The $y = 0$ test is written as $y \neq 0$.

Effect: If $y_o = 1$, the program loops. If $y_o \neq 0$, the output is 1. This is incorrect except when $x_o = 1$, or $x_o = -1$ and y_o is even. All subdomains are revealing.

X input classes		Y input classes	
X_1:	$x_0 < -1$	Y_1:	$y_0 = 0$
X_2:	$x_0 = -1$	Y_2:	$y_0 = 2^n$, $n \geq 1$
X_3:	$x_0 = 0$	Y_3:	$y_0 = 2^n - 1$, $n \geq 1$
X_4:	$x_0 = 1$	Y_4:	$y_0 = \frac{2}{3}(4^n - 1)$, $n \geq 2$
X_5:	$x_0 > 1$	Y_5:	$y_0 = \frac{1}{3}(4^{n+1} - 1)$, $n \geq 1$
		Y_6:	$y_0 < 0$

Subdomains: $X_i \times Y_j$, $i = 1,\ldots,5$; $j = 1,\ldots,5$

$X \times Y_6$

Figure 7. Subdomains for exponentiation program

Error 3: The test for odd y is written as a test for even y.

Effect: Since this error effectively reverses the Yes and No exits of the test, the result will be to compute x_o^w, where w is the 1's complement of the binary representation of y_o. The program's output will be $x_o^{(2^k-1-y_o)}$ where $k = \lfloor \log_2 y_o \rfloor + 1$, or the number of bits in y_o's binary representation. This is the wrong answer unless $y_o = 0$; all subdomains are revealing for this error.

Error 4: Failure to test for y odd; all loop iterations go through the $z \leftarrow z * x$ calculation.

Effect: Incorrect result produced except when $y_o = 0$, $x_o = 0$, $x_o = 1$, $x_o = -1$ and y_o odd, or $y_o = 2^n - 1$. Subdomains $Y_5 \times X_2$, $Y_1 \times$ all X_i, $Y_3 \times$ all X_i are all processed correctly; all others are processed incorrectly. All subdomains are revealing.

Two other errors for which every subdomain is revealing are E_5: Writing the $z \leftarrow z * x$ statement as $z \leftarrow z + x$, and E_6: Writing $y \leftarrow y \div 2$ as $y \leftarrow y-2$.

The error of writing $x \leftarrow x * x$ as $x \leftarrow x + x$ is not revealed by every one of the 26 subdomains. For inputs $(0,y_o)$, $(x_o,1)$, and for the pairs $(x_o,y_o) = (2,2)$ and $(2,3)$ the program would produce the correct result. Because of the last three cases the subdomains $(X_1, X_2, X_4, X_5) \times Y_3$, $X_5 \times Y_2$, and $X_5 \times Y_3$ are not revealing. All the other subdomains are revealing, however, and in most of these the wrong answer is produced. The error will certainly be detected.

Example 3: The program in Figure 8, taken from Geller [4], is supposed to find the number of days between two dates in the same calendar year. The input is two pairs (month1, day1) and (month2, day2), and a year. It is assumed that the first date precedes the second, and that the inputs will always be valid in all other respect.

Despite two proofs of correctness in [4], the program as it appears there has both syntactic and logical errors. We have corrected the syntactic errors, but left the logical errors to demonstrate the use of revealing subdomains. In deriving these subdomains we omit forming explicitly the path domain and problem partitions.

The input domain divides very naturally, based on the structure of the input, on the expected output, and on observations about the program's treatment of inputs. The subdomains are as follows:

S_1: the set of date pairs such that month1 = month2.

Since the program explicitly tests for this condition, it is clear that there is significance to this subdomain.

For the remaining subdomains we must consider dates in different months; there are various ways to make the divisions, but consideration of whether or not a leap year calculation is involved will play a part in any partitioning.

S_2: month1 > 2 and month2 > month1 + 1

Here there is no leap year calculation and at least one intervening month.

S_3: month1 $\neq$ 2 and month2 = month1 + 1

Again there is no leap year calculation, but also no intervening months.

S_4: month1 = 1 and month2 > 2

S_5: month1 = 2 and month2 > 2

```
procedure calendar (integer value day1, month1, day2, month2, year);
  begin
    integer days;
    if month2 = month1 then days := day2 - day1
      comment if the dates are in the same month, we can compute
              the number of days between them immediately;
    else
      begin
        integer array daysin (1: 12);
        daysin(1) := 31;  daysin(3) := 31;  daysin(4) := 30;
        daysin(5) := 31;  daysin(6) := 30;  daysin(7) := 31;
        daysin(8) := 31;  daysin(9) := 30;  daysin(10) := 31;
        daysin(11) := 30;  daysin(12) := 31;
        if ((year rem 4) = 0) or
           ((year rem 100) = 0 and (year rem 400) = 0)
           then daysin(2) := 28
           else daysin(2) := 29;
        comment set daysin(2) according to whether or not
                year is a leap year;
        days := day2 + (daysin(month1) - day1);
        comment this gives (the correct number of days -
                days in complete intervening months);
        for I := month1 + 1 until month2 - 1 do
          days := daysin(I) + days;
        comment add in the days in complete intervening months;
      end;
    write(days)
  end;
```

Figure 8. Calendar computation program

S_4 and S_5 contain inputs which require leap year calculations. To account for the different ways in which a year can be a leap year, we further divide S_4 and S_5 into four subclasses:

Y_1: year rem 4 = 0 and year rem 100 $\neq$ 0

Y_2: year rem 100 = 0 and year rem 400 $\neq$ 0

Y_3: year rem 400 = 0

Y_4: year rem 4 $\neq$ 0

The final set of subdomains is thus S_1, S_2, S_3, $S_4 \cap Y_i$, and $S_5 \cap Y_i$, $i = 1,2,3,4$ The most obvious types of errors for the calendar program are "off-by-one" errors. We might expect the program's output to be the correct value ± 1, or to be off by one month's worth of days.

Note that all the subdomains are pairwise disjoint, and that their union is the problem's entire domain. If we can show that each subdomain is revealing for a given set of errors, and then run successful tests on elements from each one, we shall have showed that the program does not contain the specified errors.

Since the result is to be the difference between two dates, the correct result for elements in S_1 is of the form day2 - day1 + k, k an integer. Since these expressions define a set of parallel lines, the correct expression is completely determined by a single input pair together with its correct output. Notice that if we were concerned about whether the proper expression should be day1 - day2 + k rather than day2 - day1 + k, then S_1 would not be revealing. These expressions are no longer nonintersecting, since when day1 = day2, (day1 - day2 + k) = (day2 - day1 + k). Thus if we feel that incorrect order of subtraction is a potential error, then S_1 must be further divided into the two subdomains:

S_1': month1 = month2 and day1 = day2

S_1'': month1 = month2 and day1 $\neq$ day2

The two new subdomains are revealing for both types of errors.

Next we note that the table of daysin is initialized correctly for the months 1,3,4,...,12. This assures that the subdomains S_2 and S_3 are revealing. S_2 and S_3 were separated since the program does additional processing when intervening months occur. Typical errors for an S_2 input are to add one too few or one too many intervening months, and these errors would occur for every S_2 input. Similarly for S_3, if an intervening month's days were incorrectly added in, the error would occur for all of S_3.

The subdomains from S_4 and S_5 reveal the errors existing in the original program. Any input from $S_4 \cap Y_i$ or $S_5 \cap Y_i$, i = 1,3,4, produces an incorrect number of days for February, and will result in an incorrect output. The inputs in $S_4 \cap Y_2$ and $S_5 \cap Y_2$ all produce correct results.

Although the most likely off-by-one errors will be revealed by all the subdomains constructed in the example, certain types of "double" off-by-one errors will not be revealed. Suppose that both indices on the for loop which adds in complete intervening months are one too high, i.e., the statement is

for I: = month1 + 2 until month2 do

Then the proper number of months will be added to the sum of days, but they will be the wrong months. If the inputs are month1 = 4, month2 = 6 the error will be detected. Month1 = 4, month2 = 7, however, will produce the correct output, for the wrong reason.

As usual, to guarantee detecting this error, a finer input partition is necessary. Because of the calendar's irregularity, a separate class is needed for each legal pair of input months. There are $\sum_{i=1}^{11} (12 - i) = 66$ such pairs. A correct version of the program contains the following replacement for the if statement in Figure 8.

```
if ((year rem 4) = 0 and (year rem 100) ≠ 0)
   or ((year rem 400) = 0)
   then daysin(2) := 29
   else daysin(2) := 28;
```

The three examples in this section were chosen to show the applicability of the revealing subdomain methodology to a range of different types of programs. They have all appeared in the testing literature, and have had diverse testing methodologies applied to them.

Our technique seems most applicable to programs whose purpose is either a classification of their inputs, or a computation based on such a classification. However, it can also be effective for programs which process their entire domain uniformly.

Since the goal of the triangle problem is to classify its inputs, it is obviously well suited to a methodology which requires a domain partition. The errors detected in this program indicate that one must not rely solely on the partition induced by the program, but must also take into account the requirements of the problem.

The calendar computation, although not directly a partitioning of the inputs, is nonetheless based on a classification of dates. The intersection of the problem and path domain partitions which was derived for the calendar problem expresses this classification. The problem is just as well suited to the revealing subdomain technique as the triangle problem.

The exponentiation problem is the example to which the methodology is the least obviously applicable; its computation does not make use of any significant partition of the input domain. However, a partition of the domain based on both the structure of the data and the program's treatment of inputs was sufficient to provide a set of test cases which effectively detected many likely errors.

7. Summary and Future Work

Although program testing has been a common practice since the first days of programming, until recently there has been almost no investigation into the theoretical basis of testing. The first efforts to construct such a theory showed the inadequacy of tests based solely on program structure, and gave conditions under which a successful test can demonstrate that a program is error-free.

However, the notion of an ideal test criterion is both program- and error-dependent, and thus does not provide a usable characterization of tests. A program independent version of an ideal criterion is unfortu tely also unusable, since no realistic criterion can possess the necessary properties.

The notion of a revealing criterion was introduced to remedy the dependence between validity and reliability, and to allow designing tests for subdomains of a problem's entire domain. A theory of testing should establish realistic and useful properties of good tests and test criteria. Defining reliability and validity in terms of a program's entire domain requires a good test to do too much. Defining good test criteria in terms of restricted subdomains allows each criterion to concentrate on likely local errors, and increases one's ability to find good tests.

Much work remains to be done in making the concepts of a revealing criterion and subdomain applicable to a wide class of problems and programs. One of the most important tasks is to find more systematic or formal methods of constructing the problem partition. This will not be easy, since finding a good problem partition is quite similar to the task of creating the program itself. Since the latter is so difficult to formalize, we may expect the former to be difficult also.

On the theoretical side, we may ask under what conditions a subdomain can be revealing for every possible error. Is this possible for any subdomain other than the trivial case of a singleton set? If we cannot achieve universally revealing, non-trivial subdomains, how close can we come? The problem is to combine realistically attainable characteristics of good tests with a reasonable level of confidence in the test results. The notion of revealing brings us a little closer to this goal.

References

1. Burstall, R. M. Program Proving as Hand Simulation with a Little Induction, Proc. IFIP Congress 74, Stockholm, North-Holland, 1974, 308-312.

2. DeMillo, R. A. , R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer, Computer, Vol. 11, No. 4, April 1978, 34-41.

3. Dijkstra, E. W. Notes on Structured Programming, in Structured Programming, ed. O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare, Academic Press, 1972, 1-81.

4. Geller, M. Test Data as an Aid in Proving Program Correctness, Comm. ACM, Vol. 21, No. 5, May 1978, 368-375.

5. Goodenough, J. B. and S. L. Gerhart, Toward a Theory of Testing: Data Selection Criteria, in Current Trends in Programming Methodology Vol. 2, ed. R. T. Yeh, Prentice-Hall, 1977, 44-79.

6. Howden, W. E. An Evaluation of the Effectiveness of Symbolic Testing, Software--Practice and Experience, Vol. 8, 1978, 381-397.

7. Howden, W. E., Reliability of the Path Analysis Testing Strategy, IEEE Trans. on Software Eng., Vol SE-2, Sept. 1976, 208-215.

8. Howden, W. .E. Symbolic Testing and the DISSECT Symbolic Evaluation System, IEEE Trans. Software Eng, Vol. SE-3, July 1977, 266-278.

9. Keirstead, R.E. and D.B. Parker. On the Feasability of Formal Certification, in Program Test Methods, ed. W. Hetzel, Prentice-Hall, 1973, 291-301.

10. Manna, Z. Mathematical Theory of Computation, McGraw-Hill, 1974.

11. Ramamoorthy, C.V., S.F. Ho, and W.T. Chen. On the Automated Generation of Program Test Data, IEEE Trans. Software Eng., Vol SE-2, Dec. 1976, 293-300.

12. Weyuker, E.J. Program Schemas with Semantic Restrictions, Ph.D. Thesis, Dept. Comp. Sci. Tech. Report DCS-TR-60, Rutgers University, New Brunswick, N.J., June 1977.